

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS
BEFORE COMPLETING FORM

2

1. REPORT NUMBER

12. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

Ada Compiler Validation Summary Report:
BiiN, BiiN Ada, Version 2.00, BiiN 60 (8 processor
configuration) (host & target), 881208W1.10010

5. TYPE OF REPORT & PERIOD COVERED
6 Dec. 1988 - 1 Dec. 1990

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

Wright-Patterson AFB
Dayton, OH, USA

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION AND ADDRESS

Wright-Patterson AFB
Dayton, OH, USA

10. PROGRAM ELEMENT, PROJECT, TASK
AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS

Ada Joint Program Office
United States Department of Defense
Washington, DC 20301-3081

12. REPORT DATE
6 December 198813. NUMBER OF PAGES
55 p.

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

Wright-Patterson AFB
Dayton, OH, USA

15. SECURITY CLASS (of this report)
UNCLASSIFIED15a. DECLASSIFICATION/DOWNGRADING
SCHEDULE
N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20 if different from Report)

UNCLASSIFIED

DTIC
ELECTE
MAY 25 1989
S D

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada
Compiler Validation Capability, ACVC, Validation Testing, Ada
Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-
1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

BiiN Ada, Version 2.00, BiiN, Wright-Patterson AFB, BiiN 60 (8 processor configuration)
under BiiN/OS, Version 1.02 (host and target), ACVC 1.10

DD FORM 1473
1 JAN 73EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

033

AD-A208 766

AVF Control Number: AVF-VSR-213.0289
88-05-20-NTL

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 881208W1.10010
BiIN
BiIN Ada, Version 2.00
BiIN 60 (8 processor configuration) Host/Target

Completion of On-Site Testing:
December 6, 1988

Prepared By:
Ada Validation Facility

ASD/SCEL Wright-Patterson AFB OH 45433-6503

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

Ada Compiler Validation Summary Report:

Compiler Name: BiIN Ada, Version 2.00

Certificate Number: 881208W1.10010

BiIN 60 (8 processor configuration) under BiIN/OS, Version 1.02

Testing Completed December 6, 1988 Using ACVC 1.10

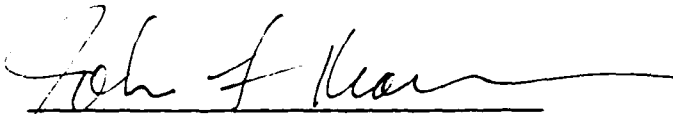
This report has been reviewed and is approved.



Ada Validation Facility

Steven P. Wilson

ASD/SCEL Wright-Patterson AFB OH 45433-6503



Ada Validation Organization

Dr. John F. Kramer

Institute for Defense Analyses

Alexandria VA 22311



Ada Joint Program Office

Dr. John Solomond

Director

Washington D.C. 20301

Ada Compiler Validation Summary Report:

Compiler Name: BiIN Ada, Version 2.00

Certificate Number: 881208W1.10010

BiIN 60 (8 processor configuration) under BiIN/OS, Version 1.02

Testing Completed December 6, 1988 Using ACVC 1.10

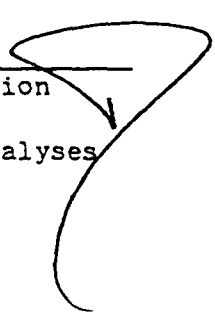
This report has been reviewed and is approved.



Ada Validation Facility

Steven P. Wilson

ASD/SCSL Wright-Patterson AFB OH 45433-6503



Ada Validation Organization

Dr. John F. Kramer

Institute for Defense Analyses

Alexandria VA 22311

Ada Joint Program Office

Dr. John Solomond

Director

Washington D.C. 20301

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	PURPOSE OF THIS VALIDATION SUMMARY REPORT	1-1
1.2	USE OF THIS VALIDATION SUMMARY REPORT	1-2
1.3	REFERENCES	1-3
1.4	DEFINITION OF TERMS	1-3
1.5	ACVC TEST CLASSES	1-4
CHAPTER 2	CONFIGURATION INFORMATION	
2.1	CONFIGURATION TESTED	2-1
2.2	IMPLEMENTATION CHARACTERISTICS	2-1
CHAPTER 3	TEST INFORMATION	
3.1	TEST RESULTS	3-1
3.2	SUMMARY OF TEST RESULTS BY CLASS	3-1
3.3	SUMMARY OF TEST RESULTS BY CHAPTER	3-2
3.4	WITHDRAWN TESTS	3-2
3.5	INAPPLICABLE TESTS	3-2
3.6	TEST, PROCESSING, AND EVALUATION MODIFICATIONS . .	3-6
3.7	ADDITIONAL TESTING INFORMATION	3-f
3.7.1	Prevalidation	3-6
3.7.2	Test Method	3-6
3.7.3	Test Site	3-7
APPENDIX A	DECLARATION OF CONFORMANCE	
APPENDIX B	APPENDIX F OF THE Ada STANDARD	
APPENDIX C	TEST PARAMETERS	
APPENDIX D	WITHDRAWN TESTS	

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent, but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- . To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- . To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- . To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed December 6, 1988 at Hillsboro OR.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
 Ada Joint Program Office
 OUSDRE
 The Pentagon, Rm 3D-139 (Fern Street)
 Washington DC 20301-3081

or from:

Ada Validation Facility
 ASD/SCEL
 Wright-Patterson AFB OH 45433-6503

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
 Institute for Defense Analyses
 1801 North Beauregard Street
 Alexandria VA 22311

1.3 REFERENCES

Reference Manual for the Ada Programming Language,
ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

Ada Compiler Validation Procedures and Guidelines, Ada Joint
Program Office, 1 January 1987.

Ada Compiler Validation Capability Implementers' Guide, SofTech,
Inc., December 1986.

Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the <u>Ada Compiler Validation Procedures and Guidelines</u> .
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.

Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer which executes the code generated by the compiler.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation and execution of legal Ada programs with certain language constructs which cannot be verified at run time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be

customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: BiiN Ada, Version 2.00

ACVC Version: 1.10

Certificate Number: 881208W1.10010

Host/Target Computer:

Machine: BiiN 60 (8 processor configuration)

Operating System: BiiN/OS, Version 1.02

Memory Size: 64 Megabytes

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

- (1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)
- (2) The compiler correctly processes tests containing loop statements nested to 63 levels. (See tests D55A03A..H (8 tests).)
- (3) The compiler rejects tests containing block statements nested to 65 levels. (See test D56001B.)
- (4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

- (1) This implementation supports the additional predefined types `SHORT_INTEGER`, `BYTE_INTEGER`, and `SHORT_FLOAT` in the package `STANDARD`. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

- (1) Some of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)
- (2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)
- (3) This implementation uses all extra bits for extra precision and uses no extra bits for extra range. (See test C35903A.)
- (4) `CONSTRAINT_ERROR` is raised for a comparison either outside the range of the predefined type `INTEGER` or greater than `SYSTEM.MAX_INT`, and no exception is raised for a membership test of the above. Also, no exception is raised when a literal operand in a comparison is outside the range of the integer type's base type. (See test C45232A.)

- (5) No exception is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)
- (6) Underflow is gradual. (See tests C45524A..Z (26 tests).)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

- (1) The method used for rounding to integer is round to even. (See tests C46012A..Z (26 tests).)
- (2) The method used for rounding to longest integer is round to even. (See tests C46012A..Z (26 tests).)
- (3) The method used for rounding to integer in static universal real expressions is round to even. (See test C4A014A.)

e. Array types.

An implementation is allowed to raise `NUMERIC_ERROR` or `CONSTRAINT_ERROR` for an array having a `'LENGTH` that exceeds `STANDARD.INTEGER'LAST` and/or `SYSTEM.MAX_INT`. For this implementation:

- (1) Declaration of an array type or subtype declaration with more than `SYSTEM.MAX_INT` components raises no exception. (See test C36003A.)
- (2) `CONSTRAINT_ERROR` is raised when `'LENGTH` is applied to an array type with `INTEGER'LAST + 2` components. (See test C36202A.)
- (3) `CONSTRAINT_ERROR` is raised when `'LENGTH` is applied to an array type with `SYSTEM.MAX_INT + 2` components. (See test C36202B.)
- (4) A packed `BOOLEAN` array having a `'LENGTH` exceeding `INTEGER'LAST` raises `CONSTRAINT_ERROR` when the array type is declared. (See test C52103X.)
- (5) A packed two-dimensional `BOOLEAN` array with more than `INTEGER'LAST` components raises `CONSTRAINT_ERROR` when the array type is declared. (See test C52104Y.)

- (6) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises CONSTRAINT_ERROR when the array type is declared. (See test E52103Y.)
- (7) In assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)
- (8) In assigning two-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

f. Discriminated types.

- (1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

- (1) In the evaluation of a multi-dimensional aggregate, the test results indicate that the order in which choices are evaluated and index subtype checks are made depends upon the aggregate itself. (See tests C43207A and C43207B.)
- (2) In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)
- (3) CONSTRAINT_ERROR is raised before all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

- (1) The pragma INLINE is supported for functions and procedures. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

1. Generics.

- (1) Generic instantiations are rejected if a required body of the generic unit has not yet been compiled, except when the generic unit and body are in the same compilation unit. If an optional body of a generic package is compiled after an instantiation, or if the body of a generic unit is recompiled after an instantiation, then the unit containing the instantiation is made obsolete. (See tests CA2009A, CA2009C, CA2009D, CA2009F, BC3204C, BC3204D, and BC3205B..D (3 tests).)
- (2) Generic specifications and bodies can be compiled in separate compilations. (See test CA1012A.)
- (3) Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A.)
- (4) Generic library subprogram specifications and bodies can be compiled in separate compilations. (See test CA1012A.)
- (5) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

j. Input and output.

- (1) The packages SEQUENTIAL_IO and DIRECT_IO cannot be instantiated with unconstrained array types or record types with discriminants without defaults. (See tests AE2101C, EE2201D, EE2201E, AE2101H, EE2401D, and EE2401G.)
- (2) Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO, DIRECT_IO, and text files. (See tests CE2102D..E, CE2102N, CE2102P, CE2102F, CE2102I..J (2 tests), CE2102R, CE2102T, CE2102V, CE3102E and CE3102I..K (3 tests).)
- (3) RESET and DELETE operations are supported for SEQUENTIAL_IO, DIRECT_IO, and text files. (See tests CE2102G, CE2102X, CE2102K, CE2102Y, CE3102F..G (2 tests), CE3104C, CE3110A, and CE3114A.)
- (4) Overwriting to a sequential file does not truncate the file. (See test CE2208B.)
- (5) Temporary files are not given names. (See tests CE2108A, CE2108C, and CE3112A.)
- (6) More than one internal file can be associated with each external file for sequential, direct, or text files when reading only or when writing only, if the external file is not

temporary. (See tests CE2107A..E (5 tests), CE2102L, CE2110B, CE2111D, CE2107F..H (3 tests), CE2110D CE2111H, CE3111A..E (5 tests), CE3114B, and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 36 tests had been withdrawn because of test errors. The AVF determined that 362 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation.

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	127	1126	1975	15	30	46	3319
Inapplicable	2	12	342	2	4	0	362
Withdrawn	1	2	33	0	0	0	36
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER														TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14		
Passed	195	576	545	243	172	99	161	332	133	36	246	297	284	3319	
N/A	18	73	135	5	0	0	5	1	4	0	6	78	37	362	
Wdrn	0	1	0	0	0	0	0	1	0	0	1	29	4	36	
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717	

3.4 WITHDRAWN TESTS

The following 36 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

A39005G	B97102E	BC3009B	CD2A62D	CD2A63A..D	CD2A66A..D
CD2A73A..D	CD2A76A..D	CD2A81G	CD2A83G	CD2A84N..M	CD2B15C
CD50110	CD5007B	CD7105A	CD7203B	CD7204B	CD7205C..D
CE2107I	CE3111C	CE3301A	CE3411B		

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 362 tests were inapplicable for the reasons indicated:

- a. C24113H..K (4 tests) are not applicable because the length of the numeric literal exceeds the allowable line length.
- b. The following 201 tests are not applicable because they have floating-point type declarations requiring more digits than `SYSTEM.MAX_DIGITS`:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

- c. C35702B and B86001U are not applicable because this implementation supports no predefined type LONG_FLOAT.
- d. The following 16 tests are not applicable because this implementation does not support a predefined type LONG_INTEGER:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45614C	C45631C
C45632C	B52004D	C55B07A	B55B09C	B86001W
CD7101F				

- e. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.
- f. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.
- g. C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because the value of SYSTEM.MAX_MANTISSA is less than 48.
- h. D55A03H uses 65 levels of loop nesting which exceeds the capacity of the compiler.
- i. D56001B uses 65 levels of block nesting which exceeds the capacity of the compiler.
- j. C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. This test recompiles package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.
- k. C96005B is not applicable because there are no values of type DURATION'BASE that are outside the range of DURATION.
- l. CA2009A, CA2009C, CA2009D, CA2009F, BC3204C, BC3009C, BC3204D, BC3205B, BC3205C, and BC3205D are not applicable because this implementation rejects a generic instantiation if a required body of the generic unit has not been compiled. If a generic unit's body is compiled or recompiled after an instantiation, the unit containing the instantiation is made obsolete.

- m. CD1009C, CD2A24I, CD2A24J, CD2A41A, CD2A41B, CD2A41E, and CD2A42A..J (10 tests) are not applicable because length clauses for floating point types are not supported by this implementation.
- n. CD2B15B is not applicable because this implementation allocates more memory to collection size than is asked for by the test.
- o. C34006D, CD2A22A, CD2A22E, CD2A22F, CD2A22I, CD2A22J, CD2A24A, CD2A24B, CD2A24E, CD2A24F, CD2A52A, CD2A52B, CD2A52G..J (4 tests), CD2A54A, CD2A54B, CD2A54G..J (4 tests), CD2A61C, CD2A61F..K (6 tests), CD2A62C, CD2A64C, CD2A65C, CD2A71A..D (4 tests), CD2A72A..D (4 tests), CD2A74A..D (4 tests), CD2A75A..D (4 tests), and CD2A84B..L (10 tests) are not applicable because of ambiguities in the LRM on 'SIZE clauses.
- p. CD5012J, CD5013S, and CD5014S are not applicable because this implementation does not support 'ADDRESS clauses for tasks.
- q. AE2101C, EE2201D, and EE2201E use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.
- r. AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.
- s. CE2102D is inapplicable because this implementation supports CREATE with IN_FILE mode for SEQUENTIAL_IO.
- t. CE2102E is inapplicable because this implementation supports CREATE with OUT_FILE mode for SEQUENTIAL_IO.
- u. CE2102F is inapplicable because this implementation supports CREATE with INOUT_FILE mode for DIRECT_IO.
- v. CE2102I is inapplicable because this implementation supports CREATE with IN_FILE mode for DIRECT_IO.
- w. CE2102J is inapplicable because this implementation supports CREATE with OUT_FILE mode for DIRECT_IO.
- x. CE2102N is inapplicable because this implementation supports OPEN with IN_FILE mode for SEQUENTIAL_IO.
- y. CE2102O is inapplicable because this implementation supports RESET with IN_FILE mode for SEQUENTIAL_IO.
- z. CE2102P is inapplicable because this implementation supports OPEN with OUT_FILE mode for SEQUENTIAL_IO.

- aa. CE2102Q is inapplicable because this implementation supports RESET with OUT_FILE mode for SEQUENTIAL_IO.
- ab. CE2102R is inapplicable because this implementation supports OPEN with INOUT_FILE mode for DIRECT_IO.
- ac. CE2102S is inapplicable because this implementation supports RESET with INOUT_FILE mode for DIRECT_IO.
- ad. CE2102T is inapplicable because this implementation supports OPEN with IN_FILE mode for DIRECT_IO.
- ae. CE2102U is inapplicable because this implementation supports RESET with IN_FILE mode for DIRECT_IO.
- af. CE2102V is inapplicable because this implementation supports OPEN with OUT_FILE mode for DIRECT_IO.
- ag. CE2102W is inapplicable because this implementation supports RESET with OUT_FILE mode for DIRECT_IO.
- ah. CE2107C..D (2 tests) and CE2107L are not applicable because multiple internal files cannot be associated with the same external file for temporary sequential files, when one or more of the internal files is writing. The proper exception is raised when multiple access is attempted.
- ai. CE2107H and CE2111H are not applicable because multiple internal files cannot be associated with the same external file for temporary direct files, when one or more of the internal files is writing. The proper exception is raised when multiple access is attempted.
- aj. CE2108B, CE2108D, and CE3112B are inapplicable because temporary files have no names.
- ak. CE3102E is inapplicable because text file CREATE with IN_FILE mode is supported by this implementation.
- al. CE3102F is inapplicable because text file RESET is supported by this implementation.
- am. CE3102G is inapplicable because text file deletion of an external file is supported by this implementation.
- an. CE3102I is inapplicable because text file CREATE with OUT_FILE mode is supported by this implementation.
- ao. CE3102J is inapplicable because text file OPEN with IN_FILE mode is supported by this implementation.

- ap. CE3102K is inapplicable because text file OPEN with OUT_FILE mode is supported by this implementation.
- aq. CE3111B and CE3115A are not applicable because multiple internal files cannot be associated with the same external file for temporary text files, when one or more of the internal files has write access. The proper exception is raised when multiple access is attempted.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that was not anticipated by the test (such as raising one exception instead of another).

No modifications were required for this validation.

On a multi-processing unit, however, tests C94020A and C95040D are erroneous in that they do not synchronize calls to the REPORT procedure COMMENT when the calls are by multiple tasks. Therefore, the comments produced during the execution of the multiple tasks were garbled. Upon the requests of the validation team, these two tests were re-run on a single processor so that an ungarbled version of the test output would be available for review for informational purposes.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the BiIN Ada, Version 2.00 compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the BiIN Ada, Version 2.00 compiler using ACVC Version 1.10 was

conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host/Target computer:	BiIN 60 (8 processor configuration)
Host/Target operating system:	BiIN/OS, Version 1.02
Compiler:	BiIN Ada, Version 2.00

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded onto a VAX 8550 and transferred by ftp to the BiIN 60 (8 processor configuration) computer. After the test files were loaded to disk, the full set of tests was compiled, linked, and all executable tests were run on the BiIN 60. Results were transferred from the BiIN 60 computer to a VAX 8550 by way of ftp for printing.

The compiler was tested using command scripts provided by BiIN and reviewed by the validation team. The compiler was tested using all default option settings except for the following:

<u>OPTION</u>	<u>EFFECT</u>
:suppress_message	Supresses display of compiler diagnostic messages for notes and warnings. Error messages are still displayed. (Default is to display all compiler messages.)
:list=source	Produces a source listing merged with error messages. (Default is to produce only an error listing.)

Tests were compiled, linked, and executed (as appropriate) using a single computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at Hillsboro OR and was completed on December 6, 1988.

APPENDIX A

DECLARATION OF CONFORMANCE

BiN has submitted the following Declaration of Conformance concerning the BiN Ada, Version 2.00 compiler.

DECLARATION OF CONFORMANCE

Compiler Implementer: BiIN™
Ada Validation Facility: ASD/SCCL, Wright-Patterson AFB OH 45433-6503
ACVC Version: 1.10

Base Configuration

Base Compiler Name:	Ada	Version:	V2.00
Host Architecture:	BiIN™ 60	OS& Version:	BiIN/OS™ V1.02
Target Architecture:	BiIN™ 60	OS& Version:	BiIN/OS™ V1.02

Implementer's Declaration

I, the undersigned, representing BiIN™, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler listed in this declaration. I declare that BiIN™ is the owner of record of the Ada language compiler listed above and, as such, is responsible for maintaining said compiler in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for the Ada language compiler listed in this declaration shall be made only in the owner's corporate name.


_____, February 24, 1989
Anastasia Czerniakiewicz, Ada Project Manager

Owner's Declaration

I, the undersigned, representing BiIN™, take full responsibility for implementation and maintenance of the Ada compiler listed above, and agree to the public disclosure of the final Validation Summary Report. I declare that the Ada language compiler listed, and its host/target performance are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.


_____, February 24, 1989
Hans Schwarz, Vice President Engineering, BiIN™

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the BiIN Ada, Version 2.00 compiler, as described in this Appendix, are provided by BiIN. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

package STANDARD is

...

type INTEGER is range -2147483648 .. 2147483647;
type SHORT_INTEGER is range -32768 .. 32767;
type BYTE_INTEGER is range -128 .. 127;

type FLOAT is digits 15 range -1.7977E308.. 1.7977E308;
type SHORT_FLOAT is digits 6 range -3.4E38 .. 3.4E38;

type DURATION is delta 9.76562E_4 range -2.097152E6 .. 2.097151999E6;

...

end STANDARD;

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

F

The Ada language definition allows for certain machine-dependences in a controlled manner. No machine-dependent syntax or semantic extensions or restrictions are allowed. The only allowed implementation-dependences correspond to implementation-dependent pragmas and attributes, certain machine-dependent conventions as mentioned in chapter 13, and certain allowed restrictions on representation clauses.

This appendix describes all implementation-dependent characteristics of BiiN™ Ada including:

- The form, allowed places, and effect of every implementation-dependent pragma
- The name and the type of every implementation-dependent attribute
- The specification of the package SYSTEM (see 13.7)
- The list of all restrictions on representation clauses (see 13.1)
- The interpretation of expressions that appear in address clauses, including those for interrupts (see 13.5)
- Any restriction on unchecked conversions (see 13.10.2)
- Any implementation-dependent characteristics of the input-output packages (see 14)
- Other implementation-defined characteristics.

F.1 Implementation-Dependent Pragmas

The following implementation-dependent pragmas are defined elsewhere in this manual. See Annex B for a summary of all predefined and implementation-defined pragmas.

- ACCESS_KIND (see 13.11.1)
- ADDRESS_MODE (see 10.7)
- ALLOCATE_WITH (see 13.11.3)
- BIND (see 13.11.2)
- EXTERNAL (see 7.7.5)
- EXTERNAL_NAME (see 13.9.2)
- HARDWARE_TYPE (see 13.11.4)
- OUTERFACE (see 13.9.1)
- PACKAGE_TYPE (see 7.7.1)
- PACKAGE_VALUE (see 7.7.2)
- POSITION_INDEPENDENT (see 7.7.6)
- PROTECTED_CALL (see 6.8.6)

- PROTECTED_RETURN (see 6.8.7)
- RESERVE_REGISTER (see 13.8.1.2)
- RETAIN_REGISTER (see 6.8.8)
- STATIC_ELABORATION (see 10.5.1)
- SUBPROGRAM_VALUE (see 6.8.2)

F.2 Implementation-Dependent Attributes

The following implementation-dependent attributes are defined elsewhere in this manual. See Annex A for a summary of all of the predefined and implementation-defined attributes.

- LABEL_OPERAND (see 13.8.1.3)
- OPERAND (see 13.8.1.3)
- PACKAGE_VALUE (see 7.7.3)
- SUBPROGRAM_OPERAND (see 13.8.1.3)
- SUBPROGRAM_VALUE. (see 6.8.3)

F.3 The Specification of the Package System

This section outlines the package SYSTEM including the implementation-dependent characteristics. This package includes the system name, the system-dependent named numbers, the definition of the type ADDRESS, the definition of UNTYPED_WORD, the definition of SUBPROGRAM_TYPE, and the definition of the ORDINAL types.

```

package SYSTEM is
  pragma STATIC_ELABORATION;

  type NAME is (TBD);
  SYSTEM_NAME: constant NAME := TBD;

  STORAGE_UNIT: constant := 8;
  MEMORY_SIZE: constant := 2**51;
  MIN_INT: constant := -2**31;
  MAX_INT: constant := 2**31 - 1;
  MAX_MANTISSA: constant := 31;
  MAX_DIGITS: constant := 15;
  FINE_DELTA: constant := 2**(-31);
  TICK: constant := 10**(-6);

  type BYTE_ORDINAL is range 0 .. 2**8 - 1;
  type SHORT_ORDINAL is range 0 .. 2**16 - 1;
  type ORDINAL is range 0 .. 2**32 - 1;
  -- The predefined operators for the BYTE_ORDINAL,
  -- SHORT_ORDINAL, and ORDINAL types are defined
  -- in this package. For the complete set of
  -- operators see Appendix F. Ordinal arithmetic
  -- operators use the machine's ordinal
  -- instructions which do not raise a CONSTRAINT
  -- error if an overflow occurs.

  type UNTYPED_WORD is private;
  -- a 32-bit type that must be word aligned.
  -- objects of this type can contain data or access values.

  type ADDRESS is
    record
      OFFSET: ORDINAL;
      AD: UNTYPED_WORD;
    end record;

```

```

and record;

function VIRTUAL (linear_address: ORDINAL)
return ADDRESS;

type SUBPROGRAM_TYPE is
record
ENTRY_NUMBER: ORDINAL;
DOMAIN_AD: UNTYPED_WORD;
and record;

for SUBPROGRAM_TYPE use
record aligned mod 32;
ENTRY_NUMBER at 0 range 0 .. 31;
DOMAIN_AD at 4 range 0 .. 31;
and record;

type HEAP_OBJECT is limited private;
type HEAP is access HEAP_OBJECT;
pragma ACCESS_KIND(HEAP, AD);

NULL_WORD: constant UNTYPED_WORD;
NULL_ADDRESS: constant ADDRESS := (0, NULL_WORD);
NULL_SUBPROGRAM: constant SUBPROGRAM_TYPE := (0, NULL_WORD);
NULL_OFFSET: constant ORDINAL := 16#F00000000#;

type EXCEPTION_TYPE is private;
function CURRENT_EXCEPTION return EXCEPTION_TYPE;

CONSTRAINT_ERROR_VALUE: constant EXCEPTION_TYPE := -- implementation defined
NUMERIC_ERROR_VALUE: constant EXCEPTION_TYPE := -- implementation defined
PROGRAM_ERROR_VALUE: constant EXCEPTION_TYPE := -- implementation defined
STORAGE_ERROR_VALUE: constant EXCEPTION_TYPE := -- implementation defined
TASKING_ERROR_VALUE: constant EXCEPTION_TYPE := -- implementation defined

private

type HEAP_OBJECT is implementation_defined;

type EXCEPTION_TYPE is implementation_defined;

end SYSTEM;

```

F.4 Restrictions on Representation Clauses

This section describes where to find the implementation restrictions for the following representation clauses:

- Pragma PACK (Section 13.1)
- Size Specification (Section 13.2)
- Alignment and Component Clauses (Section 13.4)
- Address Clauses (Section 13.5)

F.5 Restrictions on Unchecked Conversions

The implementation places no restrictions on the use of the generic function `UNCHECKED_CONVERSION` (see 13.10).

F.6 Implementation-Dependent Characteristics of Input-Output Packages

For sequential and direct files:

- The size of the created file is Operating System dependent.
- BiIN™ Ada fully supports deletion of external files.
- BiIN™ Ada fully supports resetting external files to the specified mode.
- BiIN™ Ada does not support alternative specification of the name parameter.

The package declarations of Sequential_IO and Direct_IO include the following implementation defined declaration:

```
type COUNT is range 0 .. INTEGER'LAST;
```

F.7 Other Implementation-Dependent Characteristics

This section describes other pertinent implementation-dependent characteristics, including the definition of a main program, the implementation limits of the compiler, and the ranges of the numeric type attributes.

F.7.1 Definition of a Main Program

Only library units that are parameterless procedures can be main programs.

F.7.2 Implementation Limits

Table F-1. Implementation Limits

Maximum	Description
120	Characters in an identifier
120	Characters in a source line
64,000	Source lines in a file
No Limit	Formal parameters in a subprogram declaration
$2^{31}-8$	Bits in any object
No Limit	Discriminators in a record type
$2^{31}-8$	Size of a statically sized record (in STORAGE_UNITS)
No Limit	Elements of an array type
$2^{31}-8$	Size of a statically sized array (in STORAGE_UNITS)
Size of an Object	Characters in a value of type STRING
$2^{31}-1$	Enumeration literals in an enumeration type definition
$2^{32}-1$	Dependent compilation units in closure

F.7.3 Attribute Values for the Implementation-Defined Numeric Types

This section gives the attribute values for the implementation-dependent numeric types defined in the package STANDARD and the package SYSTEM, including the attribute values for integer types, ordinal types, and floating-point types.

F.7.3.1 Integer Attribute Values

Table F-2. Integer Attribute Values

Type	FIRST	LAST	SIZE
BYTE_INTEGER	-2^7	$2^7 - 1$	8
SHORT_INTEGER	-2^{15}	$2^{15} - 1$	16
INTEGER	-2^{31}	$2^{31} - 1$	32

F.7.3.2 Ordinal Attribute Values

Table F-3. Ordinal Attribute Values

Type	FIRST	LAST	SIZE
BYTE_ORDINAL	0	$2^8 - 1$	8
SHORT_ORDINAL	0	$2^{16} - 1$	16
ORDINAL	0	$2^{32} - 1$	32

F.7.3.3 Floating-Point Attribute Values

Table F-4. Floating-Point Attribute Values

Type	DIGITS	FIRST	LAST	SIZE
SHORT_FLOAT	6	$-3.4E38$	$3.4E38$	32
FLOAT	15	$-1.8E308$	$1.8E308$	64

F.4 Restrictions on Representation Clauses

Pragma Pack: (from Section 13.1)

- For a composite type, the pragma pack only affects the type's scalar components; the pragma does not transitively apply to any composite components.
- When packing an array type, the compiler reduces intercomponent gaps and minimizes the size of scalar components. The number of bits allocated to each component is made a power of two if the type size of the component is less than 8 bits; otherwise, the number of allocated bits is a multiple of eight.
- When packing a record type, the compiler only reduces the intercomponent gaps. The space allocated to each component is not minimized. (You should use a component clause to minimize component storage sizes.)

Size specification: (from Section 13.2)

- Discrete types have a maximum size of 32.
- Linear, AD, and heap_offset access types can only have a size of 32; virtual access types can only have a size of 64.
- SHORT_FLOAT types can only have a size of 32; FLOAT types can only have a size of 64.
- A length clause will pack arrays of scalar components. However, no packing is done for record types.
- The compiler appends an extra byte to the end of array objects whose components span 3 bytes. A length clause for these kind of array objects must include storage for this extra byte.

Alignment and Component clauses: (from Section 13.4)

- The static expression in an alignment clause is in units of bits. The allowed values are 8, 16, 32, 64, and 128.
- If the bit ranges in a component clause cover 25 to 32 bits, then the placement of a field by a component clause must not span across 5 bytes.
- If the bit ranges in a component clause cover more than 32 bits, then the field must start at a byte boundary and the range must be a byte multiple.
- The offset of an access type component must start at a word boundary.
- Record fields that are array types with 3-byte components will have an extra byte allocated at the end if an access type field follows the array component or the array is at the end of the object.

- Records with a 3-byte field at the end must be word-aligned if they are used as a field in another record that contains an access type component.

Address clauses: (from Section 13.5)

- In BiNTM Ada, the only use of an address clause is to specify a memory address of an object. Address clauses for packages, subprograms, tasks, and entries are not allowed. (See the BiN OSTM User's Guide for information on writing and installing interrupt handlers.)
- An address clause may not be given for a formal parameter, a loop parameter, or a name declared by a renaming declaration.

F.6 Implementation-Dependent Characteristics of Input-Output Packages

- If a null string is passed as the NAME parameter to CREATE, an unnamed (temporary) file is created that is deleted when the file is closed.
- Trailing spaces in the NAME parameter are ignored.
- On program termination, all files are closed; the order in which they are closed is undefined.
- The procedure CREATE raises USE_ERROR if an object is already stored under the name specified, such as a file or a directory.
- The procedure OPEN raises USE_ERROR if the name does not specify an object that supports the Byte_Stream_Access_Method (such as a text file or a terminal). This includes the degenerate case of an object that does not exist.
- The procedure RESET can be called with a mode which is different than the original mode of the file.
- Any number of internal files can be associated with the same external file; the implementation imposes no restrictions. The internal files can have any mode. However, the semantics of concurrently writing and reading the same external file with different internal files is undefined. The problems are the same as when multiple programs access the same external file. Semantics are well-defined when all internal files have MODE IN_FILE; if one or more files have MODE OUT_FILE, then results are undefined.
- For Text_IO, the line terminator is ASCII LF, the page terminator is ASCII FF, and the file terminator is the end of the file; no explicit character is used. If terminators are explicitly written (as a character or part of a string in a call to PUT), they assume the semantics of an implicitly written terminator. No other control characters have special semantics.
- Interactive files are buffered as lines. That is, a GET operation will not return until a line has been input (although the GET may only process part of the line) and the output buffer is flushed whenever a NEW_LINE is done or input on the same interactive file is done. The END_OF_PAGE and END_OF_FILE functions require lookahead, so these functions cause a line to be input.
- The implementation does not use any synchronization. The caller is responsible for any process/task I/O synchronization.

- The implementation uses an internal buffer as an optimization to avoid system calls for I/O. The output buffer is flushed when the buffer is full, the file is CLOSED, the file is RESET, the external file is interactive and a line is terminated or a line is input for the same external file. No operation is provided to explicitly flush the output buffer. Interactive input files are buffered a line at a time instead of filling the buffer.
- Direct_IO and Sequential_IO may not be instantiated with an unconstrained array type (such as STRING), nor with record types with a discriminant with no default initial value.

F.8 Interpretation of Expressions in Address Clauses

The type SYSTEM_ADDRESS is defined as a record type with two components:

```

type ADDRESS is
  record
    OFFSET: ORDINAL;
    AD:     UNTYPED_WORD;
  end record;

```

Object addressing on BiiNTM systems requires these two components: an access descriptor (AD) to select an object, and an offset from the beginning of the object. When used in an address clause, the given address value may be static or dynamic. In either case, it is the user's responsibility to guarantee that there is sufficient storage space at the given address for the object.

=
-

PREDEFINED LANGUAGE ATTRIBUTES

A

This annex summarizes the definitions given elsewhere of the predefined language attributes.

P' ADDRESS

For a prefix P that denotes an object, a program unit, a label, or an entry:

Yields the address of the first of the storage units allocated to P. For a subprogram, package, task unit, or label, this value refers to the machine code associated with the corresponding body or statement. For an entry for which an address clause has been given, the value refers to the corresponding hardware interrupt. The value of this attribute is of the type ADDRESS defined in the package SYSTEM. (See 13.7.2.)

P' AFT

For a prefix P that denotes a fixed point subtype:

Yields the number of decimal digits needed after the point to accommodate the precision of the subtype P, unless the delta of the subtype P is greater than 0.1, in which case the attribute yields the value one. (P' AFT is the smallest positive integer N for which $(10^{**N}) * P' DELTA$ is greater than or equal to one.) The value of this attribute is of the type *universal_integer*. (See 3.5.10.)

P' BASE

For a prefix P that denotes a type or subtype:

This attribute denotes the base type of P. It is only allowed as the prefix of the name of another attribute: for example, P' BASE' FIRST. (See 3.3.3.)

P' CALLABLE

For a prefix P that is appropriate for a task type:

Yields the value FALSE when the execution of the task P is either completed or terminated, or when the task is abnormal; yields the value TRUE otherwise. The value of this attribute is of the predefined type BOOLEAN. (See 9.9.)

P' CONSTRAINED

For a prefix P that denotes an object of a type with discriminants:

Yields the value TRUE if a discriminant constraint applies to the object P, or if the object is a constant (including a formal parameter or generic formal parameter of mode in); yields the value FALSE otherwise. If P is a generic formal parameter of mode in out, or if P is a formal parameter of mode in out or out and the type mark given in the corresponding parameter specification denotes an unconstrained type with discriminants, then the value of this attribute is obtained from that of the corresponding actual parameter. The value of this attribute is of the predefined type BOOLEAN. (See 3.7.4.)

P' CONSTRAINED

For a prefix P that denotes a private type or subtype:

Yields the value FALSE if P denotes an unconstrained nonformal private type with discriminants; also yields the value

FALSE if P denotes a generic formal private type and the associated actual subtype is either an unconstrained type with discriminants or an unconstrained array type; yields the value TRUE otherwise. The value of this attribute is of the predefined type *BOOLEAN*. (See 7.4.2.)

P' COUNT

For a prefix P that denotes an entry of a task unit:

Yields the number of entry calls presently queued on the entry (if the attribute is evaluated within an accept statement for the entry P, the count does not include the calling task). The value of this attribute is of the type *universal_integer*. (See 9.9.)

P' DELTA

For a prefix P that denotes a fixed point subtype:

Yields the value of the delta specified in the fixed accuracy definition for the subtype P. The value of this attribute is of the type *universal_real*. (See 3.5.10.)

P' DIGITS

For a prefix P that denotes a floating point subtype:

Yields the number of decimal digits in the decimal *mantissa* of model numbers of the subtype P. (This attribute yields the number D of section 3.5.7.) The value of this attribute is of the type *universal_integer*. (See 3.5.8.)

P' EMAX

For a prefix P that denotes a floating point subtype:

Yields the largest exponent value in the binary canonical form of model numbers of the subtype P. (This attribute yields the product 4*B of section 3.5.7.) The value of this attribute is of the type *universal_integer*. (See 3.5.8.)

P' EPSILON

For a prefix P that denotes a floating point subtype:

Yields the absolute value of the difference between the model number 1.0 and the next model number above, for the subtype P. The value of this attribute is of the type *universal_real*. (See 3.5.8.)

P' FIRST

For a prefix P that denotes a scalar type, or a subtype of a scalar type:

Yields the lower bound of P. The value of this attribute has the same type as P. (See 3.5.)

P' FIRST

For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:

Yields the lower bound of the index range. The value of this attribute has the same type as this lower bound. (See 3.6.2 and 3.8.2.)

P' FIRST (N)

For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:

Yields the lower bound of the N-th index range. The value of this attribute has the same type as this lower bound. The argument N must be a static expression of type *universal_integer*. The value of N must be positive (nonzero) and no greater than the dimensionality of the array. (See 3.6.2 and 3.8.2.)

P' FIRST_BIT	<p>For a prefix P that denotes a component of a record object:</p> <p>Yields the offset, from the start of the first of the storage units occupied by the component, of the first bit occupied by the component. This offset is measured in bits. The value of this attribute is of the type <i>universal_integer</i>. (See 13.7.2.)</p>
P' FORE	<p>For a prefix P that denotes a fixed point subtype:</p> <p>Yields the minimum number of characters needed for the integer part of the decimal representation of any value of the subtype P, assuming that the representation does not include an exponent, but includes a one-character prefix that is either a minus sign or a space. (This minimum number does not include superfluous zeros or underlines, and is at least two.) The value of this attribute is of the type <i>universal_integer</i>. (See 3.5.10.)</p>
P' IMAGE	<p>For a prefix P that denotes a discrete type or subtype:</p> <p>This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the predefined type <i>STRING</i>. The result is the <i>image</i> of the value of X, that is, a sequence of characters representing the value in display form. The image of an integer value is the corresponding decimal literal; without underlines, leading zeros, exponent, or trailing spaces; but with a one character prefix that is either a minus sign or a space.</p> <p>The image of an enumeration value is either the corresponding identifier in upper case or the corresponding character literal (including the two apostrophes); neither leading nor trailing spaces are included. The image of a character other than a graphic character is implementation-defined. (See 3.5.5.)</p>
P' LABEL_OPERAND	<p>For a prefix P that denotes a label defined within a <i>MACHINE_CODE</i> procedure:</p> <p>Yields a value used for displacements in a code statement. The value of this attribute is of type <i>INTEGER</i>. (See 13.8.1.3.)</p>
P' LARGE	<p>For a prefix P that denotes a real subtype:</p> <p>The attribute yields the largest positive model number of the subtype P. The value of this attribute is of the type <i>universal_real</i>. (See 3.5.8 and 3.5.10.)</p>
P' LAST	<p>For a prefix P that denotes a scalar type, or a subtype of a scalar type:</p> <p>Yields the upper bound of P. The value of this attribute has the same type as P. (See 3.5.)</p>
P' LAST	<p>For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:</p> <p>Yields the upper bound of the first index range. The value of this attribute has the same type as this upper bound. (See 3.6.2 and 2.8.2.)</p>
P' LAST (N)	<p>For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:</p>

	<p>Yields the upper bound of the N-th index range. The value of this attribute has the same type as this upper bound. The argument N must be a static expression of type <i>universal_integer</i>. The value of N must be positive (nonzero) and no greater than the dimensionality of the array. (See 3.6.2 and 3.8.2.)</p>
P' LAST_BIT	<p>For a prefix P that denotes a component of a record object:</p> <p>Yields the offset, from the start of the first of the storage units occupied by the component, of the last bit occupied by the component. This offset is measured in bits. The value of this attribute is of the type <i>universal_integer</i>. (See 13.7.2.)</p>
P' LENGTH	<p>For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:</p> <p>Yields the number of values of the first index range (zero for a null range). The value of this attribute is of the type <i>universal_integer</i>. (See 3.6.2.)</p>
P' LENGTH (N)	<p>For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:</p> <p>Yields the number of values of the N-th index range (zero for a null range). The value of this attribute is of the type <i>universal_integer</i>. The argument N must be a static expression of type <i>universal_integer</i>. The value of N must be positive (nonzero) and no greater than the dimensionality of the array. (See 3.6.2 and 3.8.2.)</p>
P' MACHINE_EMAX	<p>For a prefix P that denotes a floating point type or subtype:</p> <p>Yields the largest value of <i>exponent</i> for the machine representation of the base type of P. The value of this attribute is of the type <i>universal_integer</i>. (See 13.7.3.)</p>
P' MACHINE_EMIN	<p>For a prefix P that denotes a floating point type or subtype:</p> <p>Yields the smallest (most negative) value of <i>exponent</i> for the machine representation of the base type of P. The value of this attribute is of the type <i>universal_integer</i>. (See 13.7.3.)</p>
P' MACHINE_MANTISSA	<p>For a prefix P that denotes a floating point type or subtype:</p> <p>Yields the number of digits in the <i>mantissa</i> for the machine representation of the base type of P (the digits are extended digits in the range 0 to P' MACHINE_RADIX - 1). The value of this attribute is of the type <i>universal_integer</i>. (See 13.7.3.)</p>
P' MACHINE_OVERFLOW	<p>For a prefix P that denotes a real type or subtype:</p> <p>Yields the value TRUE if every predefined operation on values of the base type of P either provides a correct result, or raises the exception NUMERIC_ERROR in overflow situations; yields the value FALSE otherwise. The value of this attribute is of the predefined type BOOLEAN. (See 13.7.3.)</p>
P' MACHINE_RADIX	<p>For a prefix P that denotes a floating point type or subtype:</p> <p>Yields the value of the <i>radix</i> used by the machine representation of the base type of P. The value of this attribute is of the type <i>universal_integer</i>. (See 13.7.3.)</p>

P' MACHINE_ROUNDS	<p>For a prefix P that denotes a real type or subtype:</p> <p>Yields the value TRUE if every predefined arithmetic operation on values of the base type of P either returns an exact result or performs rounding; yields the value FALSE otherwise. The value of this attribute is of the predefined type BOOLEAN. (See 13.7.3.)</p>
P' MANTISSA	<p>For a prefix P that denotes a real subtype:</p> <p>Yields the number of binary digits in the binary mantissa of model numbers of the subtype P. (This attribute yields the number B of section 3.5.7 for a floating point type, or of section 3.5.9 for a fixed point type.) The value of this attribute is of the type <i>universal_integer</i>. (See 3.5.8 and 3.5.10.)</p>
P' OPERAND	<p>For a prefix P that denotes a simple name or expanded name, an indexed component, a selected component, an attribute, a character literal, or a slice:</p> <p>Yields an operand value of the prefix P. The value of this attribute is of type MACHINE_CODE.OPERAND_TYPE. (See 13.8.1.3.)</p>
P' PACKAGE_VALUE	<p>For a prefix P that denotes a package unit:</p> <p>Yields the package value of the package unit. The value of this attribute is of type SYSTEM.UNTYPED_WORD. If the pragma PACKAGE_VALUE was not specified in the package P, then the value returned is SYSTEM.NULL_WORD, and a warning is generated at compile-time. (See 7.7.1.3.)</p>
P' POS	<p>For a prefix P that denotes a discrete type or subtype:</p> <p>This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the type <i>universal_integer</i>. The result is the position number of the value of the actual parameter. (See 3.5.5.)</p>
P' POSITION	<p>For a prefix P that denotes a component of a record object:</p> <p>Yields the offset, from the start of the first storage unit occupied by the record, of the first of the storage units occupied by the component. This offset is measured in storage units. The value of this attribute is of the type <i>universal_integer</i>. (See 13.7.2.)</p>
P' PRED	<p>For a prefix P that denotes a discrete type or subtype:</p> <p>This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the base type of P. The result is the value whose position number is one less than that of X. The exception CONSTRAINT_ERROR is raised if X equals P'BASE'FIRST. (See 3.5.5.)</p>
P' RANGE	<p>For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:</p> <p>Yields the first index range of P, that is, the range P'FIRST .. P'LAST. (See 3.6.2.)</p>
P' RANGE (N)	<p>For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:</p>

	Yields the N-th index range of P, that is, the range P'FIRST(N) .. P'LAST(N). (See 3.6.2.)
P' SAFE_EMAX	For a prefix P that denotes a floating point type or subtype: Yields the largest exponent value in the binary canonical form of safe numbers of the base type of P. (This attribute yields the number E of section 3.5.7.) The value of this attribute is of the type <i>universal_integer</i> . (See 3.5.8.)
P' SAFE_LARGE	For a prefix P that denotes a real type or subtype: Yields the largest positive safe number of the base type of P. The value of this attribute is of the type <i>universal_real</i> . (See 3.5.8 and 3.5.10.)
P' SAFE_SMALL	For a prefix P that denotes a real type or subtype: Yields the smallest positive (nonzero) safe number of the base type of P. The value of this attribute is of the type <i>universal_real</i> . (See 3.5.8 and 3.5.10.)
P' SIZE	For a prefix P that denotes an object: Yields the number of bits allocated to hold the object. The value of this attribute is of the type <i>universal_integer</i> . (See 13.7.2.)
P' SIZE	For a prefix P that denotes any type or subtype: Yields the minimum number of bits that is needed by the implementation to hold any possible object of the type or subtype P. The value of this attribute is of the type <i>universal_integer</i> . (See 13.7.2.)
P' SMALL	For a prefix P that denotes a real subtype: Yields the smallest positive (nonzero) model number of the subtype P. The value of this attribute is of the type <i>universal_real</i> . (See 3.5.8 and 3.5.10.)
P' STORAGE_SIZE	For a prefix P that denotes an access type or subtype: Yields the total number of storage units reserved for the collection associated with the base type of P. The value of this attribute is of the type <i>universal_integer</i> . (See 13.7.2.)
P' STORAGE_SIZE	For a prefix P that denotes a task type or a task object: Yields the number of storage units reserved for each activation of a task of the type P or for the activation of the task object P. The value of this attribute is of the type <i>universal_integer</i> . (See 13.7.2.)
P' SUBPROGRAM_OPERAND	For a prefix P that denotes a subprogram unit: Yields a subprogram operand of the prefix P. The value of this attribute is of type <i>MACHINE_CODE.MEM_OPERAND_TYPE</i> . This attribute can only be used in a code statement with the CALLX machine instruction. The subprogram cannot be for a predefined operator.
P' SUBPROGRAM_VALUE	For a prefix P that denotes a subprogram unit: Yields a value of type <i>SYSTEM.SUBPROGRAM_TYPE</i> . When P is not an external subprogram, the attribute yields the value <i>SYSTEM.NULL_SUBPROGRAM</i> . (See 6.8.3.)

P' SUCC

For a prefix P that denotes a discrete type or subtype:

This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the base type of P. The result is the value whose position number is one greater than that of X. The exception CONSTRAINT_ERROR is raised if X equals P' BASE' LAST. (See 3.5.5.)

P' TERMINATED

For a prefix P that is appropriate for a task type:

Yields the value TRUE if the task P is terminated; yields the value FALSE otherwise. The value of this attribute is of the predefined type BOOLEAN. (See 9.9.)

P' VAL

For a prefix P that denotes a discrete type or subtype:

This attribute is a special function with a single parameter X which can be of any integer type. The result type is the base type of P. The result is the value whose position number is the *universal_integer* value corresponding to X. The exception CONSTRAINT_ERROR is raised if the *universal_integer* value corresponding to X is not in the range P' POS (P' BASE' FIRST) .. P' POS (P' BASE' LAST). (See 3.5.5.)

P' VALUE

For a prefix P that denotes a discrete type or subtype:

This attribute is a function with a single parameter. The actual parameter X must be a value of the predefined type STRING. The result type is the base type of P. Any leading and any trailing spaces of the sequence of characters that corresponds to X are ignored.

For an enumeration type, if the sequence of characters has the syntax of an enumeration literal and if this literal exists for the base type of P, the result is the corresponding enumeration value. For an integer type, if the sequence of characters has the syntax of an integer literal, with an optional single leading character that is a plus or minus sign, and if there is a corresponding value in the base type of P, the result is this value. In any other case, the exception CONSTRAINT_ERROR is raised. (See 3.5.5.)

P' WIDTH

For a prefix P that denotes a discrete subtype:

Yields the maximum image length over all values of the subtype P (the *image* is the sequence of characters returned by the attribute IMAGE). The value of this attribute is of the type *universal_integer*. (See 3.5.5.)

PREDEFINED LANGUAGE PRAGMAS

B

This annex defines the pragmas `LIST`, `PAGE`, and `OPTIMIZE`, and summarizes the definitions given elsewhere of the remaining language-defined pragmas.

<i>Pragma</i>	<i>Meaning</i>
<code>ACCESS_KIND</code>	Takes as arguments the simple name of an access type, an address kind identifier and, optionally, a heap name. The address kind identifier must be <code>AD</code> , <code>HEAP_OFFSET</code> , <code>LINEAR</code> , or <code>VIRTUAL</code> . The pragma is allowed immediately within the declarative part or package specification that contains the declaration of the access type; the declaration must occur before the pragma. The pragma cannot be applied to a generic formal type. The pragma specifies the type of address used to represent values of the access type (see 13.11.1).
<code>ADDRESS_MODE</code>	Takes one of the identifiers <code>LINEAR</code> or <code>VIRTUAL</code> as the single argument. The pragma must appear prior to all compilation units, and affects all compilation units in the file (see 10.7).
<code>ALLOCATE_WITH</code>	<p>The pragma has two forms.</p> <p>The first form takes as arguments the identifier <code>AD</code>, an expression that is an access descriptor (<code>AD</code>) to a storage resource object (<code>SRO</code>), and an optional argument, an expression that is an <code>AD</code> to a type definition object (<code>TDO</code>). The pragma specifies the <code>SRO</code> and, optionally, the <code>TDO</code> to use for dynamic allocations of objects of access types of kind <code>AD</code>.</p> <p>The second form takes as arguments the identifier <code>VIRTUAL</code> and a heap expression. The heap expression must return an object of type <code>SYSTEM.HEAP_TYPE</code>. The pragma specifies the heap object to use for dynamic allocations of objects of access types of kind <code>VIRTUAL</code>.</p> <p>This pragma must appear within a declarative part, or package specification. The scope of this pragma extends from the pragma to the end of the declarative region associated with the innermost enclosing block statement or program unit. The pragma affects all allocators that occur directly in its scope, unless hidden by an inner occurrence of an <code>ALLOCATE_WITH</code> pragma, in which case the inner occurrence is used. Only one occurrence of each form of the pragma is allowed in a single declarative region (see 13.11.3).</p>
<code>BIND</code>	Takes as arguments a simple name and a non-null string literal expression. The simple name must be a constant of an access type that is not a generic formal type. The pragma is allowed immediately in the same declarative part or package specification as the simple name declaration; the declaration must occur before the pragma. The pragma allows an access type constant to be initialized with an access descriptor (<code>AD</code>) (see 13.11.2).

CONTROLLED

Takes the simple name of an access type as the single argument. This pragma is only allowed immediately within the declarative part or package specification that contains the declaration of the access type; the declaration must occur before the pragma. This pragma is not allowed for a derived type. This pragma specifies that automatic storage reclamation must not be performed for objects designated by values of the access type, except upon leaving the innermost block statement, subprogram body, or task body that encloses the access type declaration, or after leaving the main program (see 4.8).

ELABORATE

Takes one or more simple names denoting library units as arguments. This pragma is only allowed immediately after the context clause of a compilation unit (before the subsequent library unit or secondary unit). Each argument must be the simple name of a library unit mentioned by the context clause. This pragma specifies that the corresponding library unit body must be elaborated before the given compilation unit. If the given compilation unit is a subunit, the library unit body must be elaborated before the body of the ancestor library unit of the subunit (see 10.5).

EXCEPTION_VALUE

Takes as arguments an exception name and a constant simple name. The simple name must be a constant that has the same structure as `SYSTEM.ADDRESS`, a four-byte ordinal offset followed by a four-byte AD. The pragma is allowed immediately in the same declarative part or package specification as the exception name declaration; the declaration must occur before the pragma. The pragma directs the compiler to associate the exception value with the exception name (see 11.8.1).

EXTERNAL

This pragma takes no argument, and must appear inside a library-level package specification. The pragma specifies that this package specification and any units it declares may be referenced by units from domains other than the one in which this package specification is compiled (see 7.7.5).

EXTERNAL_NAME

Takes a subprogram name and a string literal as arguments. This pragma is allowed at the place of a declarative item and must apply in this case to a subprogram declared by an earlier declarative item of the same declarative part or package specification. This pragma is also allowed for a library unit; in this case the pragma must appear after the subprogram declaration, and before any subsequent compilation unit. The string literal is the subprogram name used by the linker (see 13.9.2).

HARDWARE_TYPE

Takes as arguments the simple name of an object of an access type, an expression that statically evaluates to an integer type, and an optional expression that statically evaluates to an integer. This pragma must appear inside a declarative part or package specification, and must follow the declaration of the access object, which must appear in the same declarative part or package specification. This pragma specifies the architectural type of an object (see 13.11.4).

INLINE

Takes one or more names as arguments; each name is either the name of a subprogram or the name of a generic sub-

INTERFACE

program. This pragma is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit. This pragma specifies that the subprogram bodies should be expanded inline at each call whenever possible; in the case of a generic subprogram, the pragma applies to calls of its instantiations (see 6.3.2).

Takes a language name and a subprogram name as arguments. This pragma is allowed at the place of a declarative item, and must apply in this case to a subprogram declared by an earlier declarative item of the same declarative part or package specification. This pragma is also allowed for a library unit; in this case the pragma must appear after the subprogram declaration, and before any subsequent compilation unit. This pragma specifies the other language (and thereby the calling conventions) and informs the compiler that an object module will be supplied for the corresponding subprogram (see 13.9).

LIST

Takes one of the identifiers ON or OFF as the single argument. This pragma is allowed anywhere a pragma is allowed. It specifies that listing of the compilation is to be continued or suspended until a LIST pragma with the opposite argument is given within the same compilation. The pragma itself is always listed if the compiler is producing a listing.

MEMORY_SIZE

Takes a numeric literal as the single argument. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the compilation. The effect of this pragma is to use the value of the specified numeric literal for the definition of the named number MEMORY_SIZE (see 13.7).

OPTIMIZE

Takes one of the identifiers TIME or SPACE as the single argument. This pragma is only allowed within a declarative part and it applies to the block or body enclosing the declarative part. It specifies whether time or space is the primary optimization criterion.

OUTERFACE

Takes as arguments a language name identifier and a subprogram name. This pragma is allowed at the place of a declarative item, and must apply in this case to a subprogram declared by an earlier declarative item of the same declarative part or specification. This pragma is also allowed for a library unit; in this case the pragma must appear after the subprogram declaration, and before any subsequent compilation unit. The pragma specifies the calling convention for calling a subprogram from BiiN™ Ada or another BiiN™ language (see 13.9.1).

PACK

Takes the simple name of a record or array type as the single argument. The allowed positions for this pragma, and the restrictions on the named type, are governed by the same rules as for a representation clause. The pragma specifies that storage minimization should be the main criterion when selecting the representation of the given type (see 13.1).

PACKAGE_TYPE

Takes a string literal as the single argument. The pragma is only allowed immediately within a package specification at the place of a declarative item before any other declaration.

PACKAGE_VALUE

The pragma indicates that the package may be provided with alternate package bodies (see 7.7.1).

Takes a simple name denoting a package type as the single argument. The pragma is allowed immediately within a package specification before any other declaration. The pragma indicates that the package can supply an alternative implementation for the package type specified by the simple name (see 7.7.2).

PAGE

This pragma has no argument, and is allowed anywhere a pragma is allowed. It specifies that the program text which follows the pragma should start on a new page (if the compiler is currently producing a listing).

POSITION_INDEPENDENT

The pragma has no argument, and is allowed only within the declarative part of a package. The pragma directs the compiler to produce code that will execute correctly, regardless of the address space in which the code is placed at execution time. The pragma is used for writing interrupt handlers (see 7.7.6).

PRIORITY

Takes a static expression of the predefined integer subtype **PRIORITY** as the single argument. This pragma is only allowed within the specification of a task unit or immediately within the outermost declarative part of a main program. It specifies the priority of the task (or tasks of the task type) or the priority of the main program (see 9.8).

PROTECTED_CALL

Takes one or more subprogram names as arguments. The pragma is allowed at the place of a declarative item. The pragma applies to all the named subprograms that are called between the appearance of the pragma and the end of the declarative region in which it was specified. The pragma indicates that before a call to one of the named subprograms is made, any global registers not used for parameter passing are to be cleared (see 6.8.6).

PROTECTED_RETURN

Takes as arguments one or more subprogram or generic subprogram names. The pragma is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation but before any subsequent compilation unit. The pragma indicates that before returning from the specified subprograms, the global registers not used for parameter passing are to be cleared (see 6.8.7).

RESERVE_REGISTER

Takes one or more names of type **MACHINE_CODE.REGISTER_TYPE** as arguments. The pragma is only allowed within the declarative part of a subprogram or block statement. The pragma instructs the compiler to reserve the specified registers for the user (see 13.8.1.2).

RETAIN_REGISTER

Takes one or more names of type **MACHINE_CODE.REGISTER_TYPE** as arguments. The pragma is only allowed in the declarative part of a subprogram. The pragma instructs the compiler to save the values of the named registers before the subprogram is called and to restore the original values to the registers after returning from the subprogram (see 6.8.8).

SHARED

Takes the simple name of a variable as the single argument. This pragma is allowed only for a variable declared by an object declaration and whose type is a scalar or access type; the variable declaration and the pragma must both occur (in this order) immediately within the same declarative part or package specification. This pragma specifies that every read or update of the variable is a synchronization point for that variable. An implementation must restrict the objects for which this pragma is allowed to objects for which each of direct reading and direct updating is implemented as an indivisible operation (see 9.11).

STATIC_ELABORATION

The pragma takes no arguments and is only allowed immediately within a library unit package specification. The pragma indicates that only static elaboration is desired for the package unit (see 10.5.1).

STORAGE_UNIT

Takes a numeric literal as the single argument. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the compilation. The effect of this pragma is to use the value of the specified numeric literal for the definition of the named number `STORAGE_UNIT` (see 13.7).

SUBPROGRAM_VALUE

Takes as arguments a subprogram type name and the simple name of a subprogram. The pragma is only allowed at the place of a declarative item in a package specification, or after a library unit in a compilation, but before any subsequent compilation unit. The pragma indicates that the subprogram specified can provide an alternate body for the subprogram type specified (see 6.8.2).

SUPPRESS

Takes as arguments the identifier of a check and optionally also the name of either an object, a type or subtype, a subprogram, a task unit, or a generic unit. This pragma is only allowed either immediately within a declarative part or immediately within a package specification. In the latter case, the only allowed form is with a name that denotes an entity (or several overloaded subprograms) declared immediately within the package specification. The permission to omit the given check extends from the place of the pragma to the end of the declarative region associated with the innermost enclosing block statement or program unit. For a pragma given in a package specification, the permission extends to the end of the scope of the named entity.

If the pragma includes a name, the permission to omit the given check is further restricted: it is given only for operations on the named object or on all objects of the base type of a named type or subtype; for calls of a named subprogram; for activations of tasks of the named task type; or for instantiations of the given generic unit (see 11.7).

SYSTEM_NAME

Takes an enumeration literal as the single argument. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the compilation. The effect of this pragma is to use the enumeration literal with the specified identifier for the definition of the constant `SYSTEM_NAME`. This pragma is only allowed if the specified

identifier corresponds to one of the literals of the type NAME
declared in the package SYSTEM (see 13.7).

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below:

<u>Name and Meaning</u>	<u>Value</u>
<u>\$ACC_SIZE</u> An integer literal whose value is the number of bits sufficient to hold any value of an access type.	64
<u>\$BIG_ID1</u> An identifier the size of the maximum input line length which is identical to \$BIG_ID2 except for the last character.	(1 .. 119 => 'A', 120 => '1')
<u>\$BIG_ID2</u> An identifier the size of the maximum input line length which is identical to \$BIG_ID1 except for the last character.	(1 .. 119 => 'A', 120 => '2')
<u>\$BIG_ID3</u> An identifier the size of the maximum input line length which is identical to \$BIG_ID4 except for a character near the middle.	(1 .. 59 => 'A', 60 => '3', 61 .. 120 => 'A')

<u>Name and Meaning</u>	<u>Value</u>
\$BIG_ID4 An identifier the size of the maximum input line length which is identical to \$BIG_ID3 except for a character near the middle.	(1 .. 59 => 'A', 60 => '4', 61 .. 120 => 'A')
\$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1 .. 117 => '0', 118 .. 120 => "298")
\$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1 .. 114 => '0', 115 .. 120 => "69.E01")
\$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	(1 .. 59 => 'A')
\$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	(1 .. 60 => 'A', 61 => '1')
\$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1 .. 100 => ' ')
\$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
\$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	288230376151711744
\$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8

<u>Name and Meaning</u>	<u>Value</u>
\$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	BiIN
\$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2#1.0#E-31
\$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	2147483647
\$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_TYPE
\$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_TYPE
\$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100000.0
\$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	20000000.0
\$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	5
\$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	X\$@Y?
\$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	X\$@Z*
\$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648

<u>Name and Meaning</u>	<u>Value</u>
\$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
\$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
\$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100000.0
\$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-20000000.0
\$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	0
\$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
\$MAX_DIGITS Maximum digits supported for floating-point types.	15
\$MAX_IN_LEN Maximum input line length permitted by the implementation.	120
\$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
\$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648
\$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1 .. 2 => "2:", 3 .. 117 => '0', 118 .. 120 => "11:")

<u>Name and Meaning</u>	<u>Value</u>
<p>\$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.</p>	(1 .. 3 => "16:", 4 .. 117 => '0', 118 .. 120 => "F.E")
<p>\$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.</p>	(1 => '"', 2 .. 119 => 'A', 120 => '"')
<p>\$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.</p>	-2147483648
<p>\$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.</p>	64
<p>\$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.</p>	BYTE_INTEGER
<p>\$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.</p>	BiIN
<p>\$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.</p>	16#FFFFFFFE#
<p>\$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than \$DEFAULT_MEM_SIZE. If there is no other value, then use \$DEFAULT_MEM_SIZE.</p>	288230376151711744

<u>Name and Meaning</u>	<u>Value</u>
<p>\$NEW_STOR_UNIT</p> <p>An integer literal whose value is a permitted argument for pragma STORAGE_UNIT, other than \$DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.</p>	8
<p>\$NEW_SYS_NAME</p> <p>A value of the type SYSTEM.NAME, other than \$DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.</p>	BiIN
<p>\$TASK_SIZE</p> <p>An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.</p>	64
<p>\$TICK</p> <p>A real literal whose value is SYSTEM.TICK.</p>	2:1.0:E-10

APPENDIX D
WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 36 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

- a. A39005G: This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).
- b. B97102E: This test contains an unintended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).
- c. BC3009B: This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).
- d. CD2A62D: This test wrongly requires that an array object's size be no greater than 10, although its subtype's size was specified to be 40 (line 137).
- e. CD2A63A..D, CD2A66A..D, CD2A73A..D, CD2A76A..D [16 tests]: These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.
- f. CD2A81G, CD2A83G, CD2A84M and N, and CD50110: These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86 and 96, 86 and 96, and 58, respectively).

- g. CD2B15C and CD7205C: These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.
- h. CD5007B: This test wrongly expects an implicitly declared subprogram to be at the the address that is specified for an unrelated subprogram (line 303).
- i. CD7105A: This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK -- particular instances of change may be less (line 29).
- j. CD7203B, and CD7204B: These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.
- k. CD7205D: This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.
- l. CE2107I: This test requires that objects of two similar scalar types be distinguished when read from a file -- DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid. (line 90)
- m. CE3111C: This test requires certain behavior when two files are associated with the same external file; however, this is not required by the Ada standard.
- n. CE3301A: This test contains several calls to END_OF_LINE and END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, and 136).
- o. CE3411B: This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.